Infinite Data

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

There are many choices available to the language designer when a function call is made. The choices made will have a significant effect on the language.
Your objectives for this lecture:

- Show how to implement thunks by using local state and user-defined types.
- Show how thunks can implement the call-by-need parameter-passing style.
- Show how to create virtually-infinite data-structures by using lazy evaluation.

We like call-by-value because

- it's efficient — we usually want the value soon anyway.
- it's easy to implement.

But, there's a cost...

- It can perform unnecessary computations.
- ... in fact, it could cause non-termination.

```
1 let rec foo x = foo (x + 1);;
2 let fTrue a b = a;;
3 fTrue 5 (foo 10);;
```


Consider the following code:

```
1 # let plus a b = print_string "Plus"; a+b;;
2 val plus : int -> int -> int = <fun>
3 # let foo a b = a * a * a;;
4 val foo : int -> 'a -> int = <fun>
5 # foo (plus 2 3) (plus 5 5);;
6 Plus
7 Plus
8 - : int = 125
9 #
```

What is the optimal number of times to run `plus`?

- OCaml is already CBV, so do what you normally do to get that.
- You can use functions to delay evaluation to get CBN.
  - Let `fun () ->` e be a delayed expression.
  - To extract the information, apply the delayed expression to `()`.

Running Example

```
# let foo a b =
    (a ()) * (a ()) * (a ());;
val foo : (unit -> int) -> 'a -> int = <fun>
# foo (fun () -> plus 2 3) (fun () -> plus 5 5);;
```

- How many times will you see `Plus` printed to the screen?

- We can use the local state technique to perform an optimization.

```
# type 'a status = Value of 'a
                 | Susp of (unit -> 'a);;
# let delay f =
    let status = ref (Susp f) in
    fun () -> match (!status) with
    | Value a -> a
    | Susp f -> let result = f () in
                ( status := (Value result);
                  result );;
val delay : (unit -> 'a) -> unit -> 'a = <fun>
# let force f = f ();;
```


Running Example

```
# let foo a b =
    (force a) * (force a) * (force a);;
val foo : (unit -> int) -> 'a -> int = <fun>
# foo (delay (fun () -> plus 2 3))
      (delay (fun () -> plus 5 5));;
```

- `(delay (fun () -> plus 2 3))` is called a *suspension*, or sometimes a *thunk*.
- How many times will `plus` be printed to the screen?

- Keyword `lazy` will create a suspension for us.

```
# let foo = lazy (plus 2 3);;
val foo : int Lazy.status ref =
          contents = Lazy.Delayed <fun>
```

- The `Lazy` module defines a `force` function.

```
# Lazy.force;;
- : 'a Lazy.t -> 'a = <fun>
# Lazy.force foo;;
Plus
- : int = 5
# Lazy.force foo;;
- : int = 5
```

Lazy Lists

- We can create a new list type that takes advantage of the lazy data...

```
type 'a llist = Cons of 'a * 'a llist Lazy.t | Nil
```

- To display these we can convert back to normal lists.

Convert Lazy List to OCaml List

```
# let rec ftake n llist =
  match n,llist with
  | _,Nil -> []
  | 0,_ -> []
  | _,(Cons (x,xs)) -> x :: ftake (n-1) (force xs)
val ftake : int -> 'a llist -> 'a list = <fun>
```


Infinite Lists

- OCaml will—if you ask nicely—allow you to make infinite data.

```
# let rec ones = Cons(1, lazy ones);;
val ones : int llist = Cons (1, contents = Delayed <f
# let rec numsfrom n = Cons(n, lazy (numsfrom (n+1)))
val numsfrom : int -> int llist = <fun>
```

- Note that `ones` isn't even a function.

Infinite Mapping

```
let rec lmap f llist =
  match llist with
  | Cons (x,xs) ->
       Cons (f x, lazy (lmap f (force xs)))
  | Nil -> Nil
```


Other Functions

```
# let rec lmap2 f lst1 lst2 =
  match lst1,lst2 with
  | Cons(x,xs),Cons(y,ys) ->
        Cons(f x y, lazy (lmap2 f (force xs)
                                  (force ys)));;
val lmap2 : ('a -> 'b -> 'c) -> 'a llist
            -> 'b llist -> 'c llist = <fun>
```

We also define `fhead` and `ftail`. Now watch this....

```
# let rec fib = Cons(1, lazy
                (Cons(1, lazy
                (lmap2 plus fib (ftail fib)))));;
val fib : int llist = Cons (1, contents = Delayed <fu
```


An Application....

- We can make a Newton's method approximator.
- For square root of n, $x_{i+1} = (x_i + n/x_i)/2$.

```
# let rec approx err alist =
    if abs((fhead alist) -
           (fhead (ftail alist))) < err
    then (fhead alist)
    else approx err (ftail alist);;
# let next n xi = (xi +. n /. xi) /. 2.0
# let mkSeq n =
   let rec seq = Cons(1.0, lazy (lmap (next n) seq))
   in seq;;
```

1. Write another version of `nats` (the list of natural numbers) without using a function.
2. Write a function `circular` that takes a normal list and returns an infinite circular list with the same data. (This one is a bit tricky.)

```
1 # let ott = circular [1;2;3];;
2 val ott : int llist = Cons (1, contents = Delayed
3 # ftake 10 ott;;
4 - : int list = [1; 2; 3; 1; 2; 3; 1; 2; 3; 1]
```

3. What happens if we pass `[]` to `circular`?


```
1  # let rec nats = Cons(1, lazy (lmap (plus 1) nats));;
2  val nats : int llist = Cons (1, contents = Delayed <f
3  # ftake 2 nats;;
4  Plus
5  Plus
6  - : int list = [1; 2]
7  # ftake 4 nats;;
8  Plus
9  Plus
10 - : int list = [1; 2; 3; 4]
11 # ftake 4 nats;;
12 - : int list = [1; 2; 3; 4]
```



```
1 let rec circular lst =
2     let rec result = lazy (aux lst)
3     and aux lst =
4       match lst with
5       | [] -> force result
6       | x::xs -> Cons(x, lazy (aux xs))
7     in force result;;
8 val circular : 'a list -> 'a llist = <fun>
```

This one is tricky, because we had to delay the `result` in line 2 to convince OCaml that this was a safe thing to do.


```
1 # circular [];;
2 Exception: Lazy.Undefined.
```

The *real* version of `Lazy` has an extra constructor for this case.

```
1 # let delay f =
2     let status = ref (Susp f) in
3     fun () -> match (!status) with
4     | Value a -> a
5     | Undefined -> raise (Failure "delay")
6     | Susp f -> (status := Undefined;
7                  let result = f () in
8                   ( status := (Value result);
9                     result ));;
```